The Von Neumann Model Internals –Computer Organization–

Lionel Morel

Computer Science and Information Technologies - INSA Lyon

Fall-Winter 2023-24

The micro-machine ISA (1/3)

In the first lab sessions, we'll look at a home-made processor.

- 8-bits instructions
- 8-bits signed integers only
- 2 8-bits registers, named `A` and `B`

Computation instructions with 1 or 2 operands

```
B -> A          21 -> B
B + A -> A      B xor -42 ->
                A
not B -> A      lsr A -> A
A xor 12 -> A   B - A -> A;
```

WARNING: some instructions that would seem "intuitive" are actually forbidden... eg: `A+B -> B` is incorrect ... `B+A -> B` is correct.

The micro-machine ISA (2/3)

Memory reads and writes

```
*A -> A        *A -> B
A -> *A        B -> *A
*cst -> A      *cst -> B
A -> *cst      B -> *cst
```

`*A` means: "the content of memory at the address contained in register A".

The micro-machine ISA (3/3)

Unconditional absolute branch

```
JA 42
```

continues execution at address 42.

Conditional relative branch

`JR offset`	`JR offset IFZ` *(executed if Z=1)*
`JR offset IFC` *(executed if C=1)*	`JR offset IFN` *(executed if N=1)*

The micro-machine ISA - example program

```
prog2: *21 -> A  ;; copy Mem[21] to register A
       *42 -> B  ;; copy Mem[42] to register B
       B+A -> B  ;; Compute B+A and put result
                 ;;    into register B
       B -> *43  ;; copy content of register B to Mem[43]
```

The micro-machine ISA - Instruction Encoding

Instruction encoding

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
saut relatif conditionnel	1	cond		offset signé sur 5 bits				

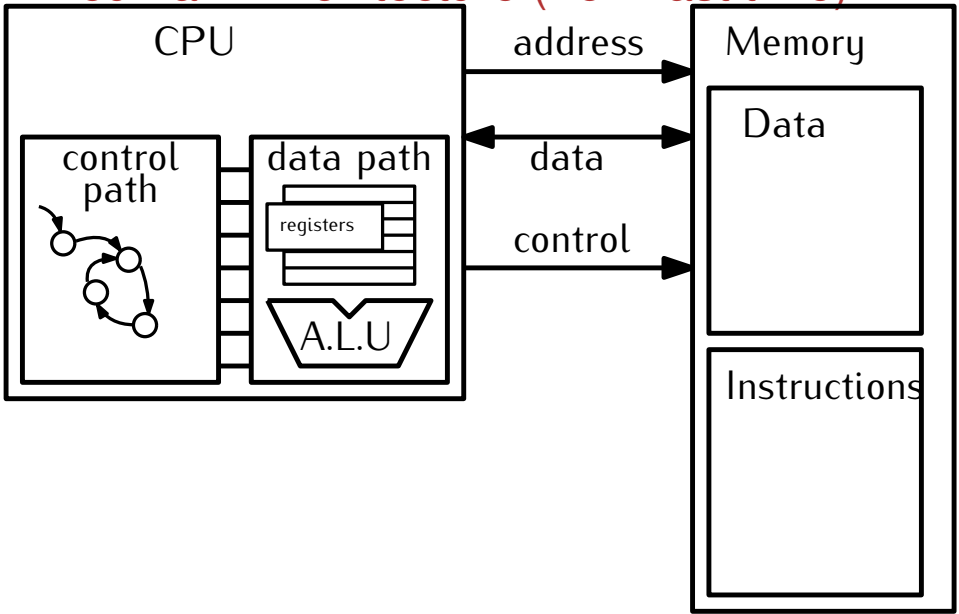
Signification des différents raccourcis utilisés

Notation	encodé par	valeurs possibles
`dest`	destS=instr[0]	`A` si destS=0, `B` si destS=1
`arg1`	arg1S=instr[1]	`A` si arg1S=0, `B` si arg1S=1
`arg2`	arg2S=instr[2]	`A` si arg2S=0, constante 8-bit si arg2S=1
`offset`	instr[4:0]	offset signé sur 5 bits

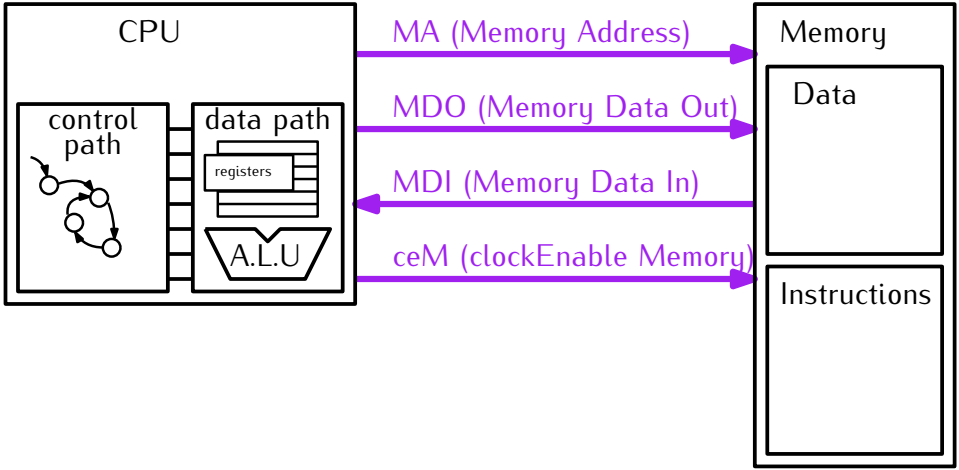
Encodage des différentes opérations possibles

codeop	mnémonique	remarques
0000	`arg1 + arg2 -> dest`	addition; shift left par `A+A->A`
0001	`arg1 - arg2 -> dest`	soustraction; `0 -> A` par `A-A->A`
0010	`arg1 and arg2 -> dest`	
0011	`arg1 or arg2 -> dest`	
0100	`arg1 xor arg2 -> dest`	
0101	`LSR arg1 -> dest`	logical shift right; bit sorti dans C; arg2 inutilisé
0110	`arg1 - arg2 ?`	comparaison arithmétique; destS inutilisé
1000	`(not) arg1 -> dest`	`not` si arg2S=1, sinon simple copie
1001	`arg2 -> dest`	arg1 inutilisé
1101	`*arg2 -> dest`	lecture mémoire; arg1S inutilisé

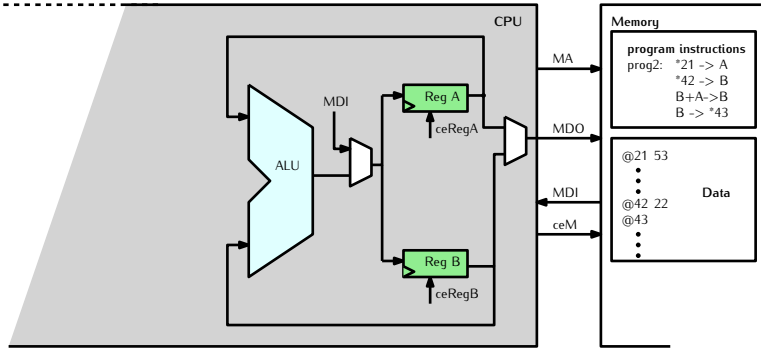
Von Neumann Architecture (from last time)



Von Neuman Architecture (refined)

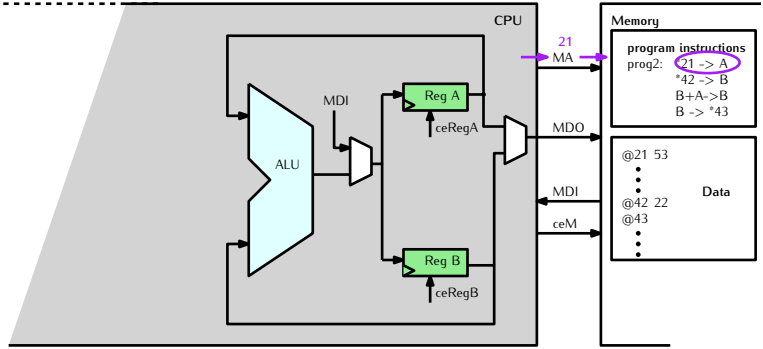


1- Execute Instructions



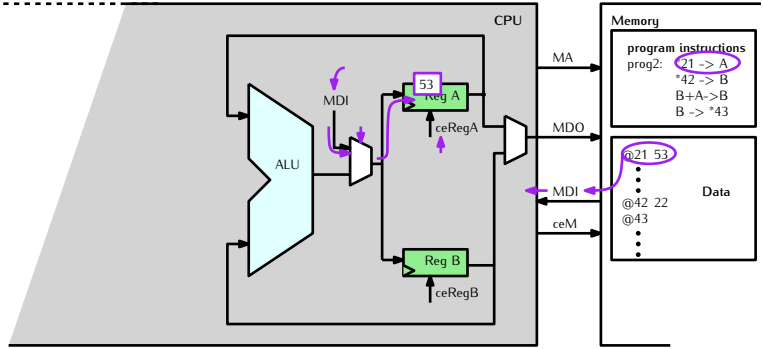
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

1- Execute Instructions



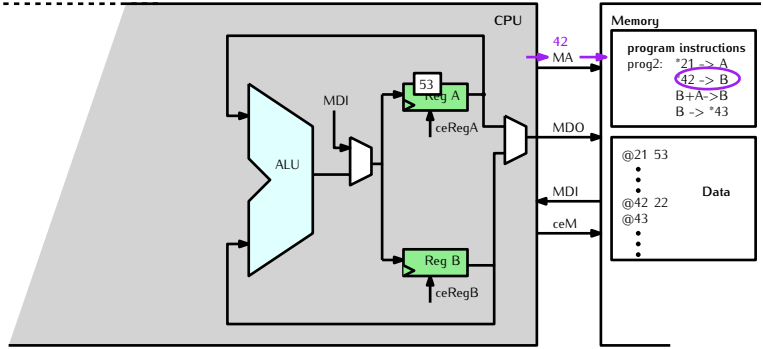
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

1- Execute Instructions



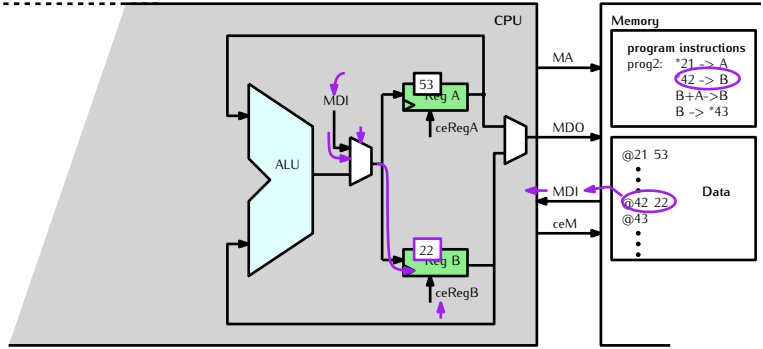
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

1- Execute Instructions



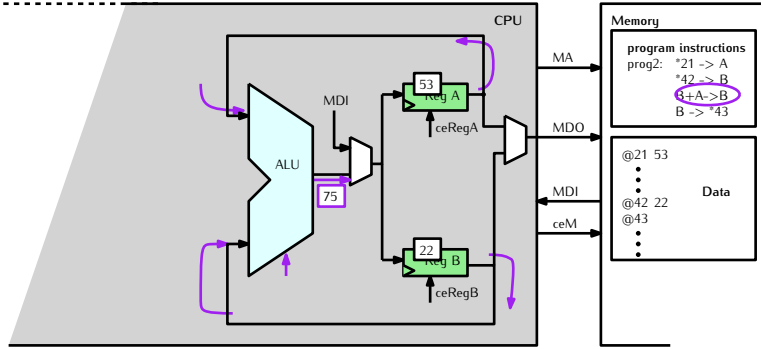
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

1- Execute Instructions



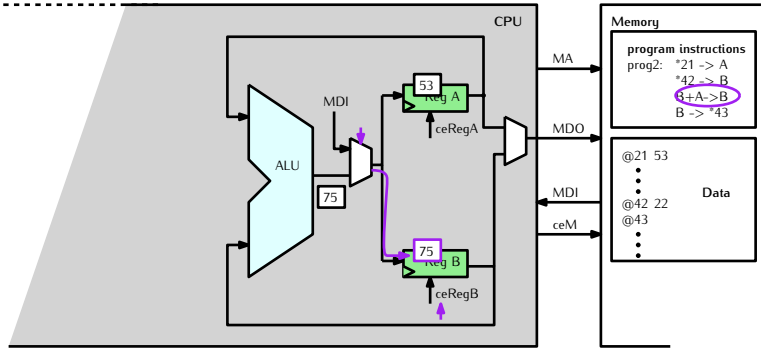
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

1- Execute Instructions



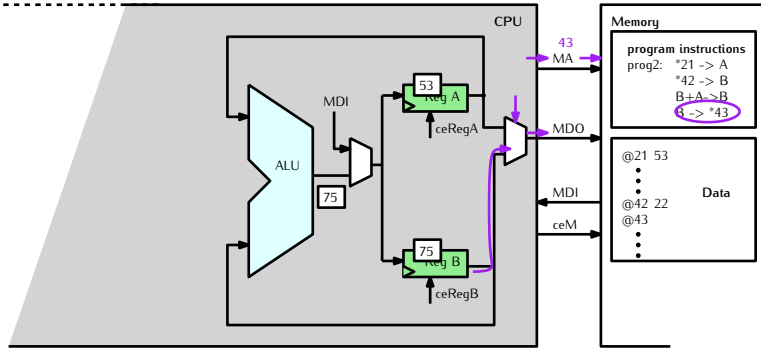
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

1- Execute Instructions



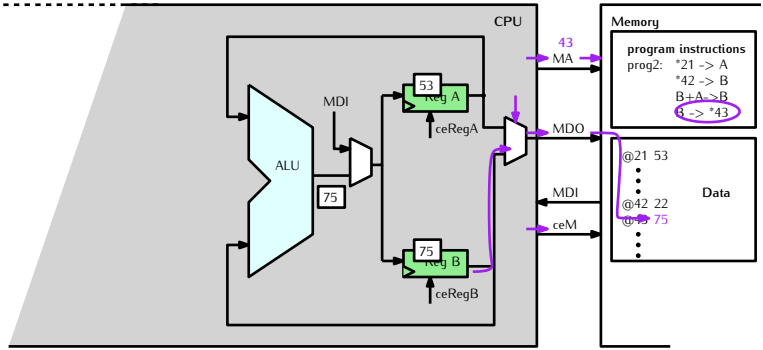
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

1- Execute Instructions



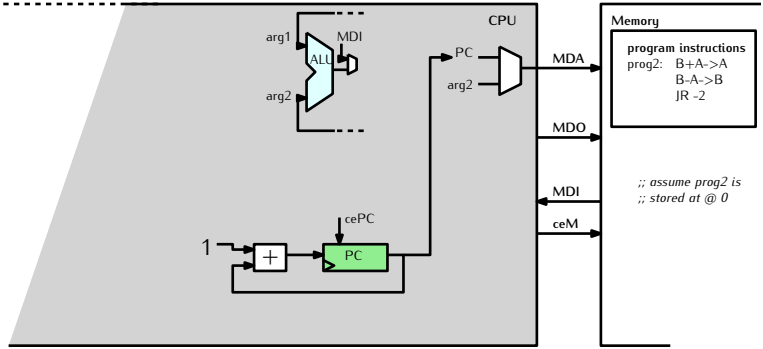
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

1- Execute Instructions



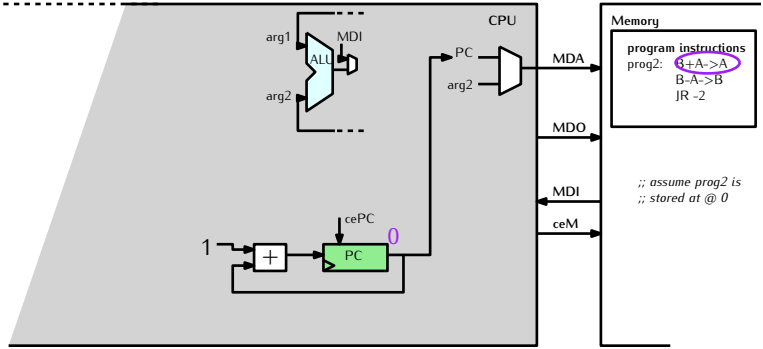
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

2- Advance in Program



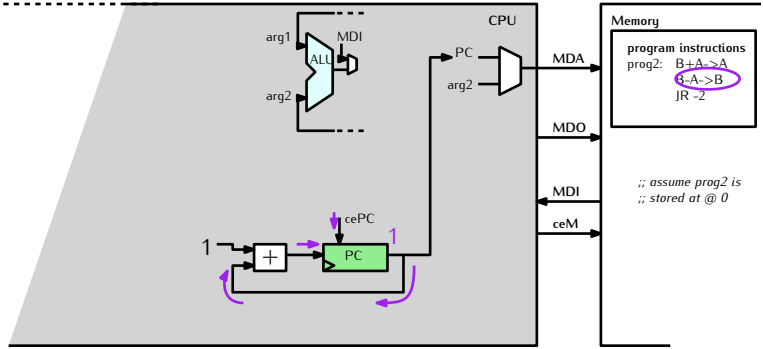
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

2- Advance in Program



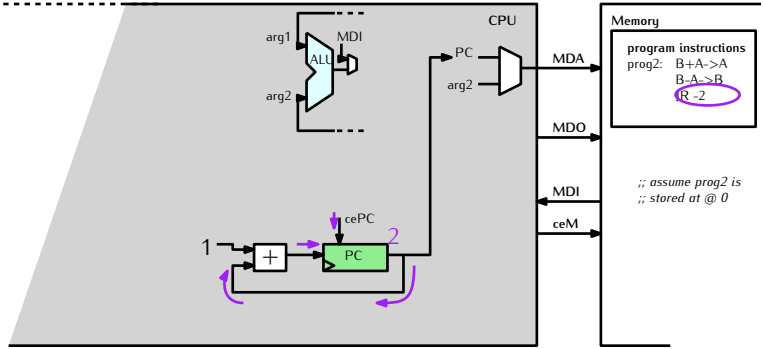
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

2- Advance in Program



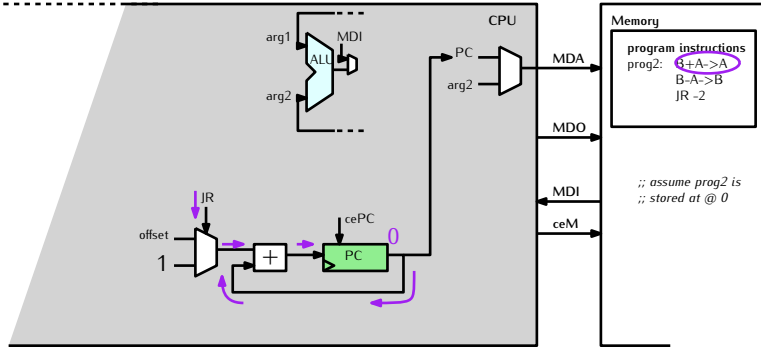
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

2- Advance in Program



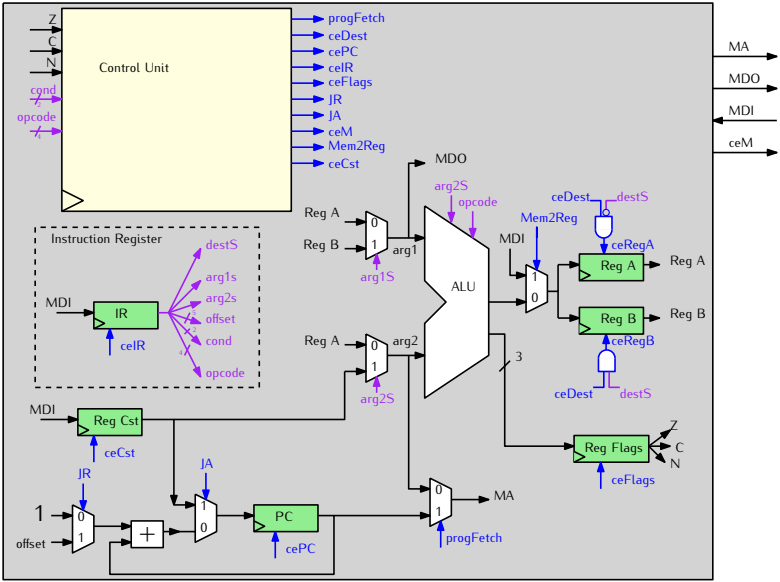
MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

2- Advance in Program

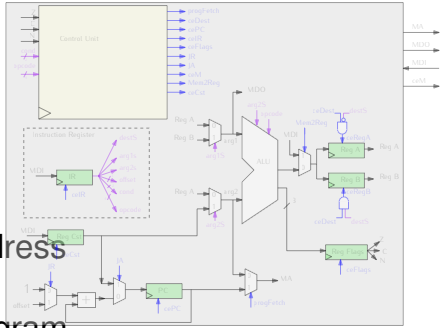


MA: Memory Address
MDO: Memory Data Output
MDI: Memory Data Input
ceM: clock-enable Memory (needs 1 when writing TO memory)

DataPath - The Big Picture

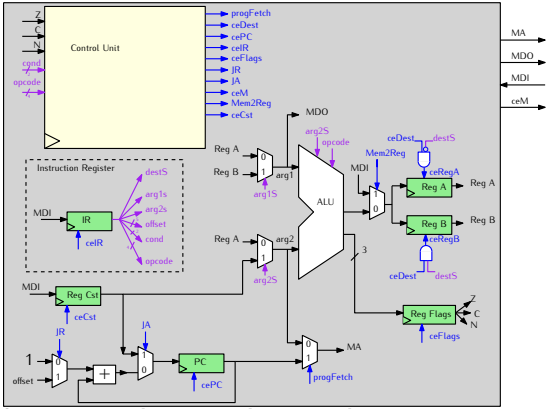


The DataPath



- Can store data for program:
  - Use General-Purpose Registers
  - Here, only 2: **RegA** and **RegB**
- Can compute on this data
- Talks to the memory through data and address buses, **MA**, **MDO** and **MDI**
- Dedicated Registers are used to track program execution:
  - **PC**: Program Counter
  - **IR**: Instruction Register
  - **Cst**: holds constants from Instructions
  - **Flags**: stores last values of ALU's flags: Z (zero), C (carry), N(egative)

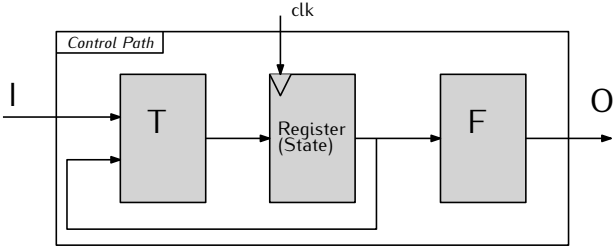
The Control Unit



- Implements the passing from one instruction to the next
- Manipulates registers through “Clock Enable” commands: **ceRegA**, **ceRegB**, **cePC**, **ceIR**, etc.
- Manipulates memory through control signals: **ceM**
- Performs selection over data to be written to registers or buses: **JR**, **JA**, **progFetch**, etc.

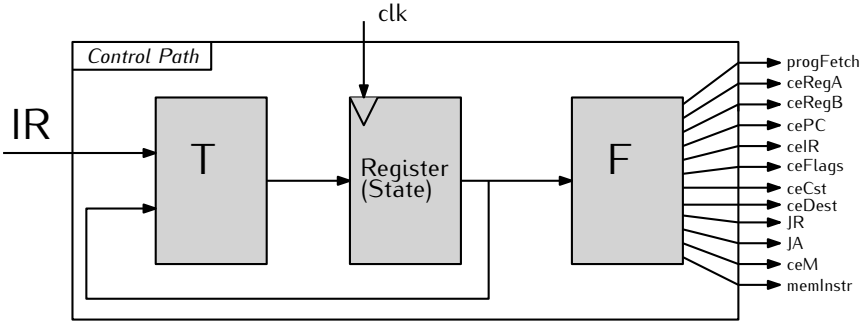
Control Unit

The Control Unit is an Algorithmic State Machine (see Guillaume Beslon's lectures)



- F is the **Output** function
- T is the **Transition** function
- F and T are **combinatorial** circuits (no registers)

Von Neumann's Control Unit - Interface



Executing Programs - The Von Neumann Cycle

Given the structure of a program in a Von Neumann's machine, the algorithm to execute it is simple:

```
Do forever:
  Fetch Instruction
  Decode Instruction
  Execute Instruction
```

A Von Neumann machine is an ASM[1] that executes this **Von Neuman Cycle** such that:

Fetch Copy the current instruction bit-vector from the memory to the processor (register IR)

Decode Look at the instruction opcode to prepare the DataPath

Execute Process the data in the DataPath such that the processor **makes the behavior of the instruction happen**

!! **Don't forget** to move to the "**next**" instruction

[1] ASM = Abstract State Machine ... different from ASM/asm for "assembler"

The Micro-machine

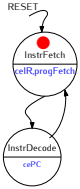
```
B -> A          21 -> B
B + A -> A      B xor -42 ->
                A
not B -> A      lsr A -> A
A xor 12 -> A   B - A -> A
B - A?
```

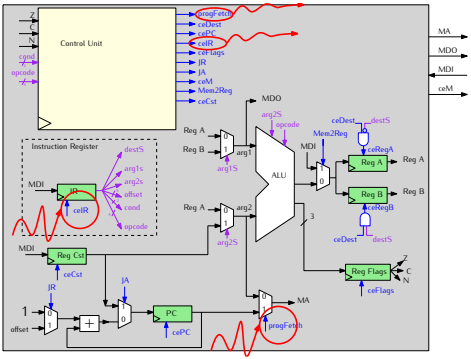
```
*A -> A         *A -> B
A -> *A         B -> *A
*cst -> A       *cst -> B
A -> *cst       B -> *cst
```

```
JA 42
```

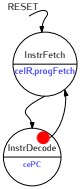
```
JR offset       JR offset IFZ
```

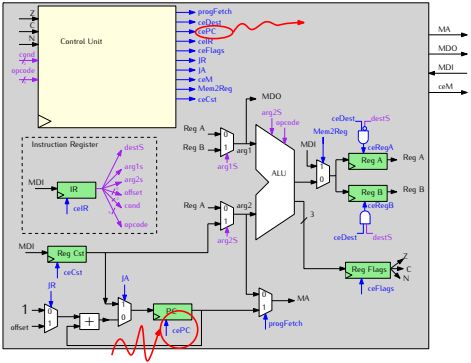
The VN's automaton - Fetch-Decode



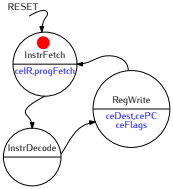


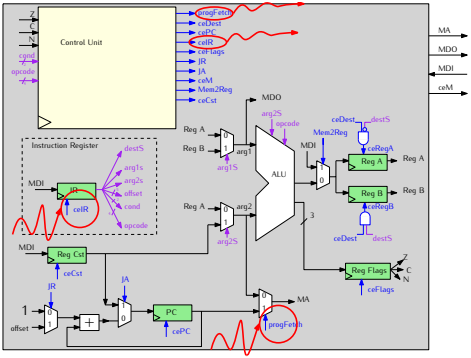
The VN's automaton - Fetch-Decode





The VN's automaton - Arith/Logic ins on 1 byte

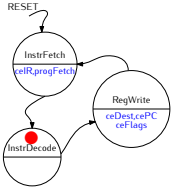


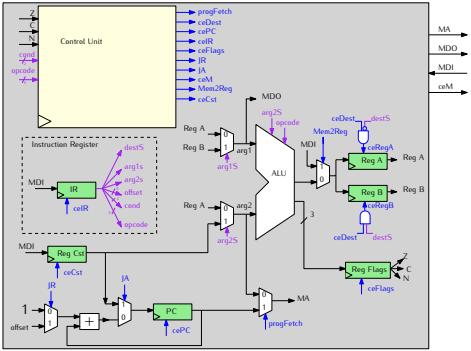


B + A -> A

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	0	0	0	0	1	0

The VN's automaton - Arith/Logic ins on 1 byte

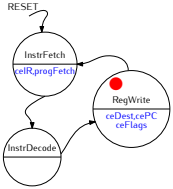


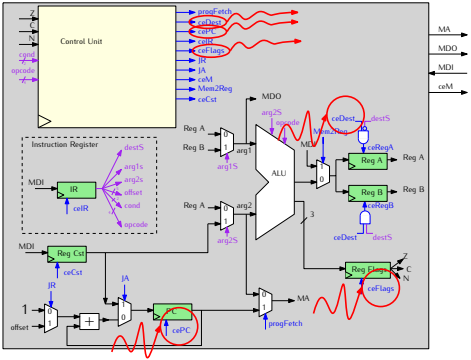



```
B + A -> A
```

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	0	0	0	0	1	0

The VN's automaton - Arith/Logic ins on 1 byte

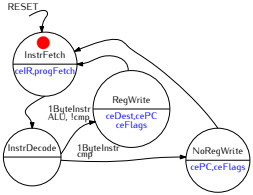


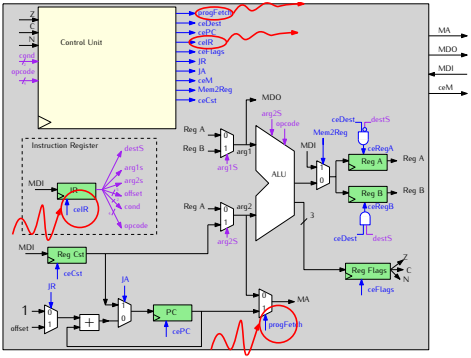



```
B + A -> A
```

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	0	0	0	0	1	0

The VN's automaton - Comparison

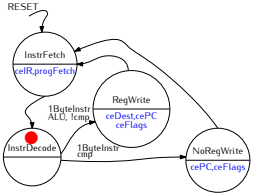


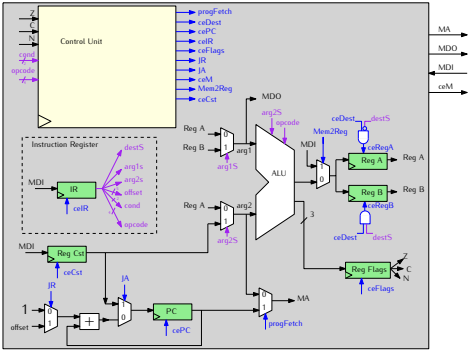



```
B - A?
```

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	0	0	0	0	1	α

The VN's automaton - Comparison

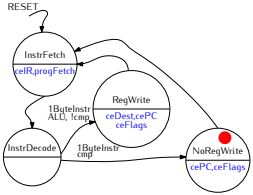


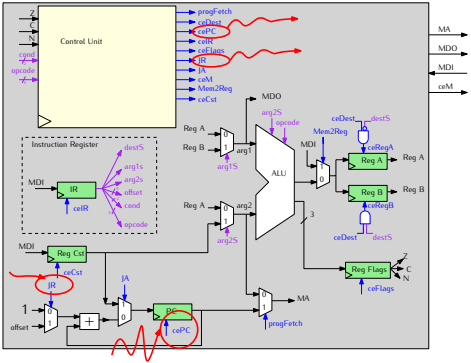


B - A?

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	0	0	0	0	1	α

The VN's automaton - Comparison

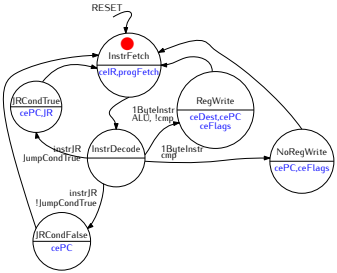


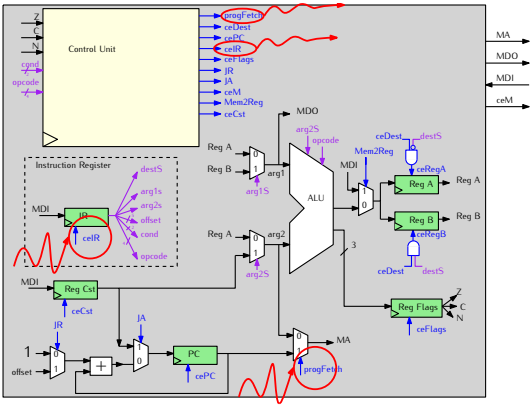


B - A?

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	0	0	0	0	1	α

The VN's automaton - Relative jumps

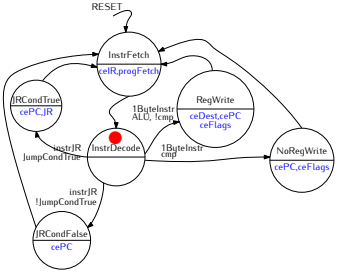


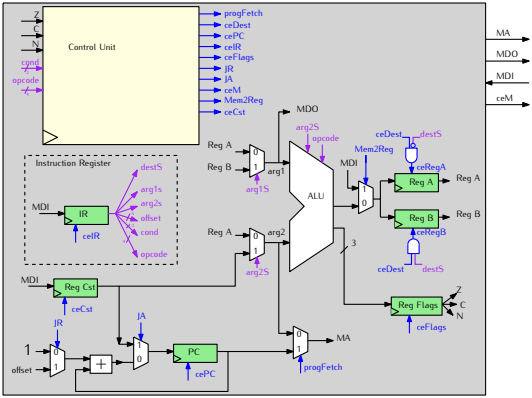



```
JR -5 IFN
```

bit	7	6	5	4	3	2	1	0
saut relatif conditionnel	1	cond		offset signé sur 5 bits				
	1	1	1	1	1	0	1	1

The VN's automaton - Relative jumps

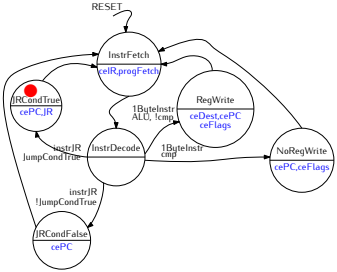


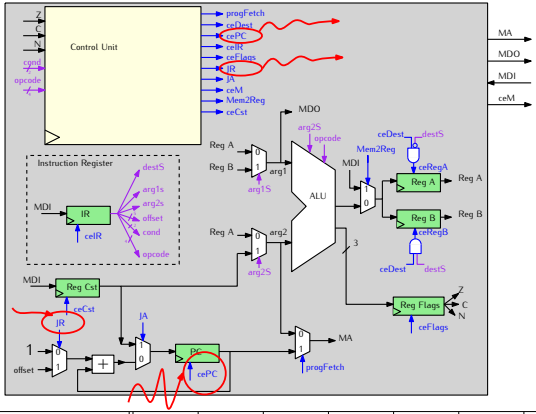



```
JR -5 IFN
```

bit	7	6	5	4	3	2	1	0
saut relatif conditionnel	1	cond		offset signé sur 5 bits				
	1	1	1	1	1	0	1	1

The VN's automaton - Relative jumps



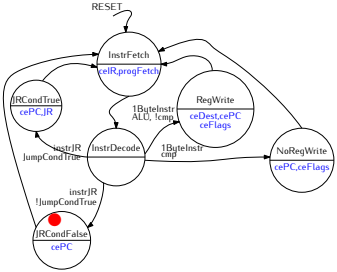


If condition is true (ie take the jump)

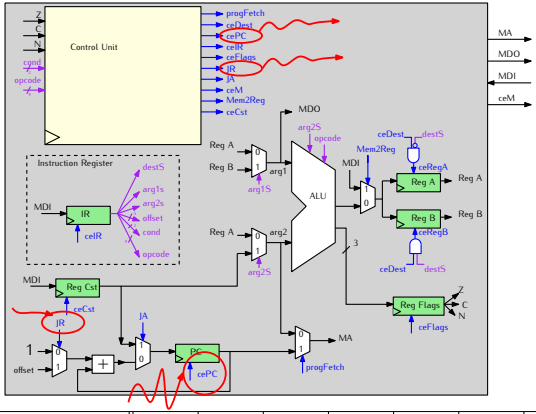
```
JR -5 IFN
```

bit	7	6	5	4	3	2	1	0
saut relatif conditionnel	1	cond		offset signé sur 5 bits				
	1	1	1	1	1	0	1	1

The VN's automaton - Relative jumps



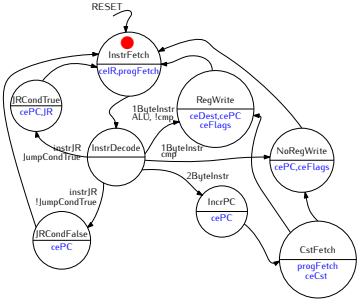
If condition is false (ie DON'T take the jump)

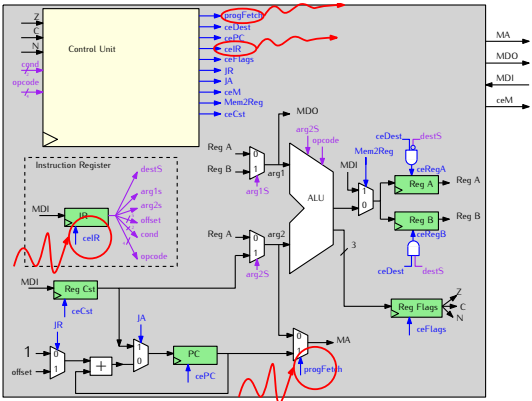



```
JR -5 IFN
```

bit	7	6	5	4	3	2	1	0
saut relatif conditionnel	1	cond		offset signé sur 5 bits				
	1	1	1	1	1	0	1	1

The VN's automaton - 2-bytes instructions

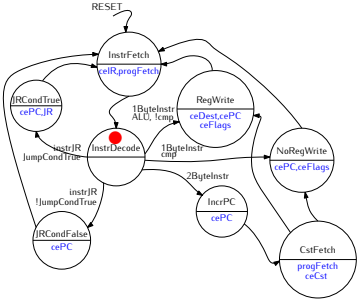


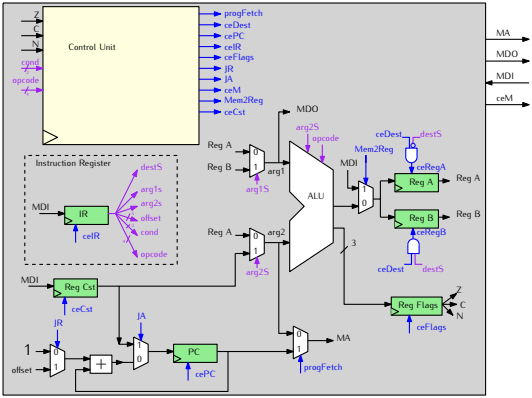


A xor 12 -> A

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	1	0	0	1	0	0
constante encodée sur 8 bits	0	0	0	0	1	1	0	0

The VN's automaton - 2-bytes instructions

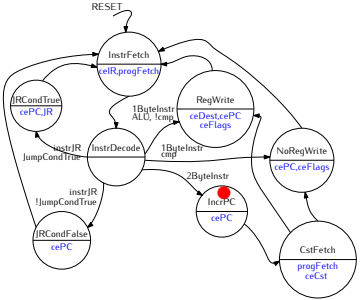


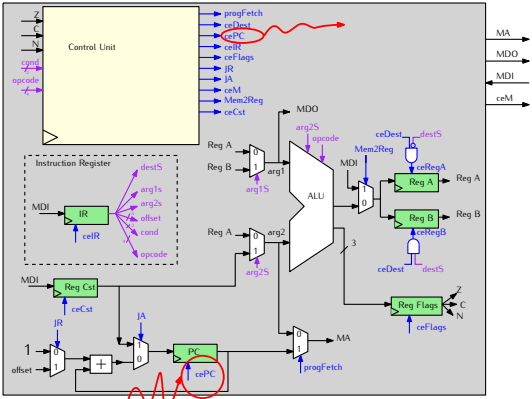



```
A xor 12 -> A
```

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	1	0	0	1	0	0
constante encodée sur 8 bits	0	0	0	0	1	1	0	0

The VN's automaton - 2-bytes instructions

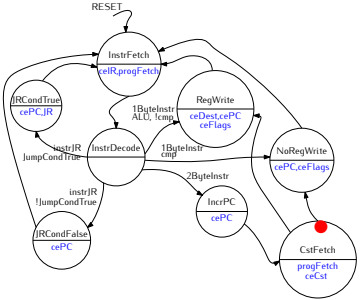


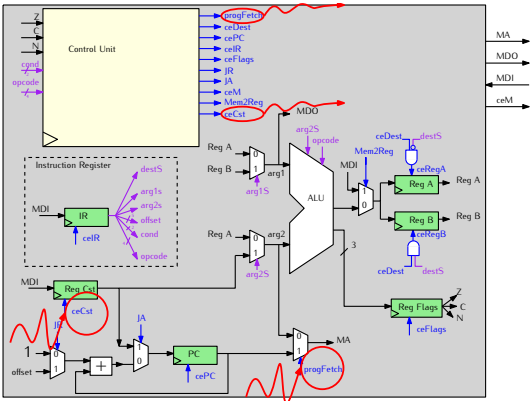



```
A xor 12 -> A
```

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	1	0	0	1	0	0
constante encodée sur 8 bits	0	0	0	0	1	1	0	0

The VN's automaton - 2-bytes instructions

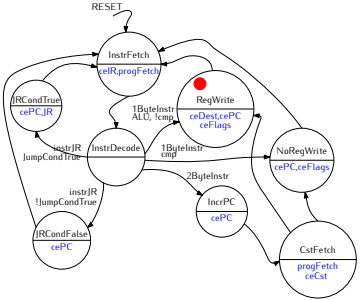


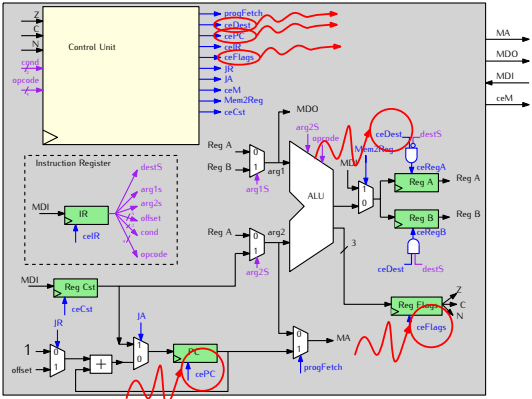


A xor 12 -> A

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	1	0	0	1	0	0
constante encodée sur 8 bits	0	0	0	0	1	1	0	0

The VN's automaton - 2-bytes instructions

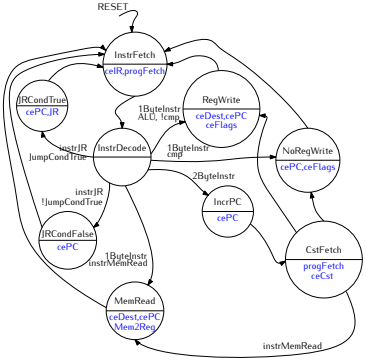


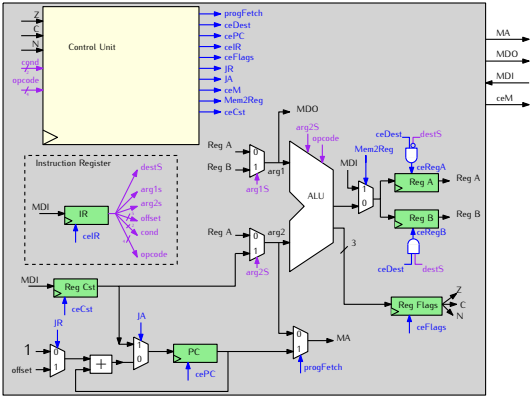



```
A xor 12 -> A
```

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	1	0	0	1	0	0
constante encodée sur 8 bits	0	0	0	0	1	1	0	0

The VN's automaton - Memory reads

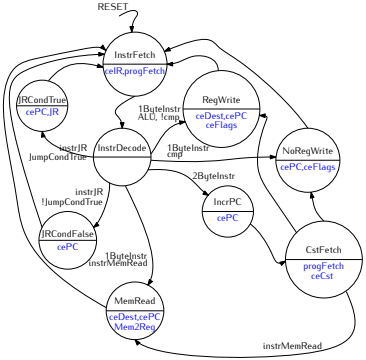


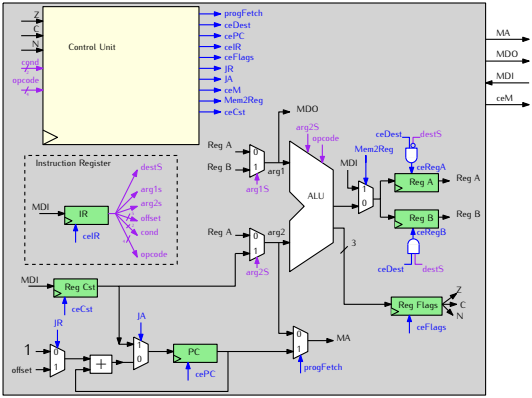



```
*A -> B
```

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	1	0	0	1	0	0

The VN's automaton - Memory reads

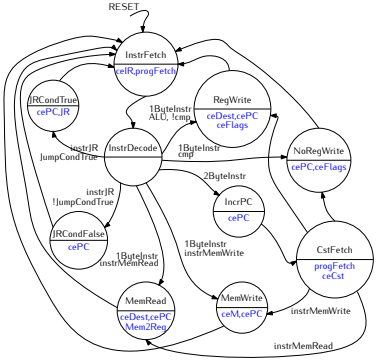


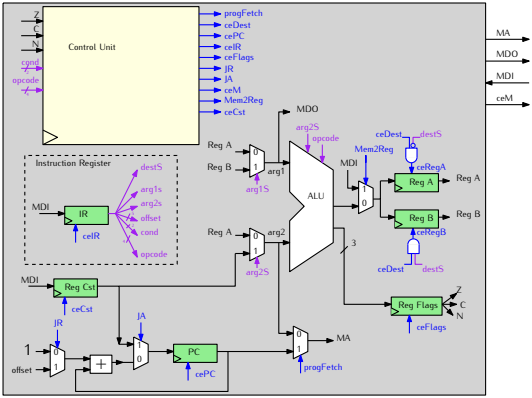


bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	1	0	0	1	0	0
constante encodée sur 8 bits	0	1	1	0	1	0	0	1

```
*105 -> B
```

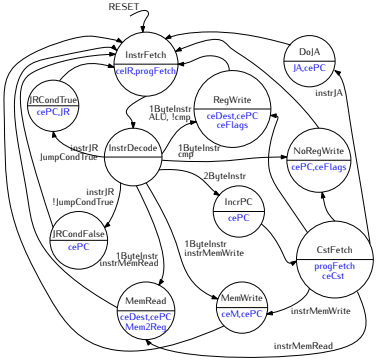
The VN's automaton - Memory writes

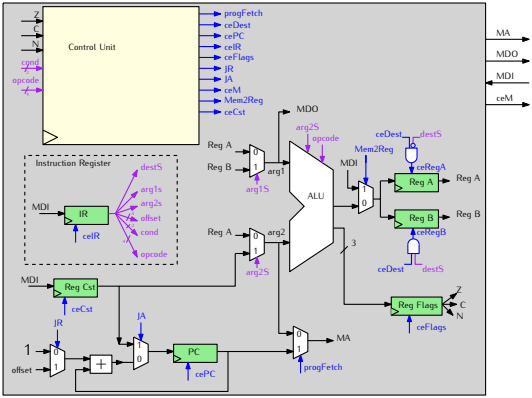





```
B -> *A
B -> *200
```

The VN's automaton - Absolute Jumps






```
JA 42
```

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop				arg2S	arg1S	destS
	0	0	1	0	0	1	0	0
constante encodée sur 8 bits	0	0	1	0	1	0	1	0

(A bit) more advanced topics

Un-Pipelined Execution

The behavior of the processor (Fetch/Decode/execute) can be refined:

IF : Instruction Fetch

- Copy instruction word from memory to IR

ID : Instruction Decode

- Prepare operands
- `JA 10`: read constant from memory
- `JR +3 IFN`: extend 5-bits +3 to 8-bits

EX : Execute

- Send correct control signal to ALU
- Let arguments run towards ALU

MEM : Write/Read stuff to/from memory

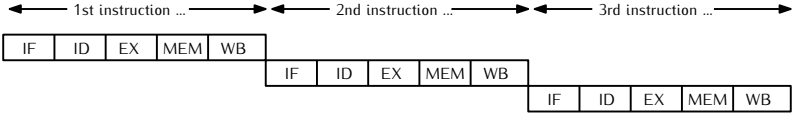
- `A -> *42`
- copy the current value of register `A` into memory at address 42.

WB : Writeback (result to register)

- Write value out of ALU into target register

Un-Pipelined Execution

Until now, the execution of *n* instructions looks like this:



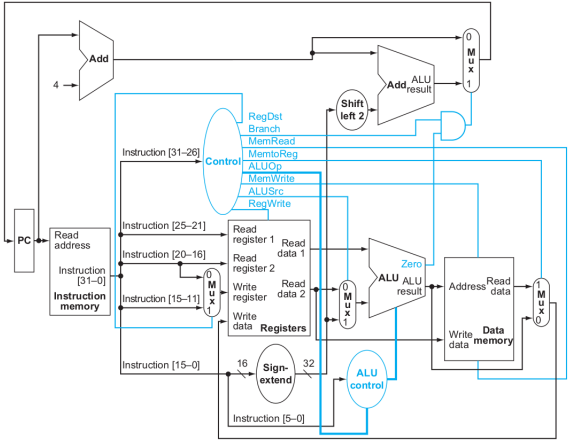
- If each step takes δns,
- Then 1 instruction takes $5 * \delta$ns,
- Then **n instructions take** $\underline{n * 5 * \delta \textbf{ns}}$ to execute (ouch!).

Pipeline - Principle

- **Steps are independent** from one another
- Each step (IF, ID, EX, MEM, WB) can be realized with a **distinct part of the datapath**
- Data can move from one stage of the datapath to the next with **no delay**

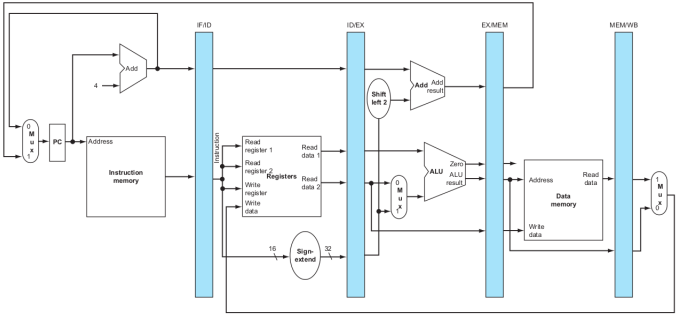
The Processor datapath can work like an **Assembly Line**

Un-pipelined versus pipelined architecture (1/2)



NB: This is taken from D. A. Patterson, & J. L. Hennessy, *Computer Organization and Design*, p265. You choose be convince yourself that our micro-machine is **very close** to this. See also `https://en.wikipedia.org/wiki/MIPS_architecture`

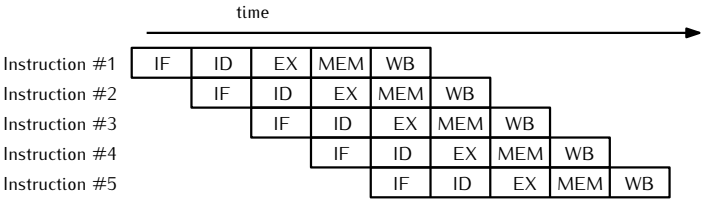
Un-pipelined versus pipelined architecture (2/2)



- Each **blue line** corresponds to a set of **stage-specific registers**
- eg IF/ID = info produced by Instruction Fetch, and needed by Instruction Decode.
- These are used to store data that needs to be passed from one stage to the next
- This is schematic and control signals are missing!!

Pipelined Execution

Now, the execution of the same n instructions looks like this:



- If each steps takes δns, then **n instructions take** $5 * \delta + (n - 1) * \delta$ = $(\mathbf{n} + \mathbf{4}) * \delta$ **ns** to execute.
- Expected gain: Processor gets D times faster (where D is the Depth of the pipeline).
- In nominal mode of course (see pipeline hazards)

Pipelined Execution

- The pipeline stages are separate parts of the CPU
- They are organized so that each stage takes one instruction at each new clock cycle
- An instruction is “passed” from one stage to the next

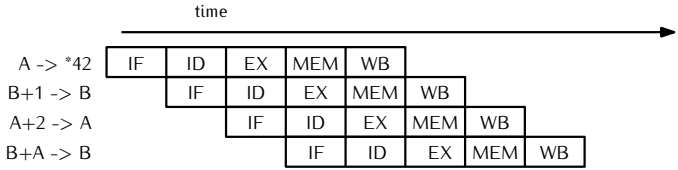
Pipeline hazards

The pipeline's behavior may be disrupted by the following:

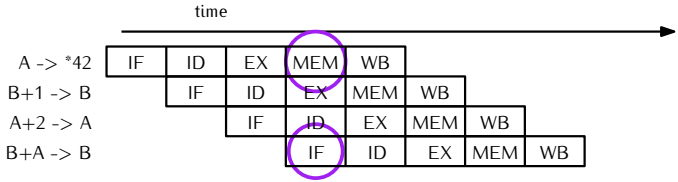
- **Structural hazard**: two instructions in different stages of the pipeline require the same resource.
- **Data hazard**: the data required by an instruction was not already produced by a preceeding instruction.
- **Control hazard**: an instruction execution is interrupted, typically by a branch.

- The general solution consists in inserting **bubbles** into the pipeline.
- This amounts to adding **NOPs**, ie "**do-nothing instructions**" into the program.

Pipeline hazards



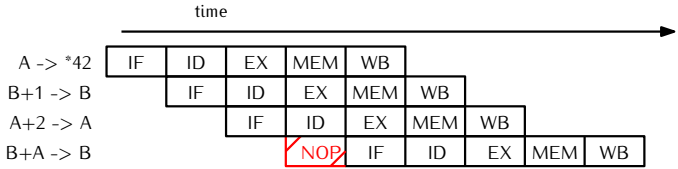
Pipeline hazards



1. Structural Hazard

Both `A->*42` and `B+A->B` require memory simultaneously. We need to **delay 2nd instruction**.

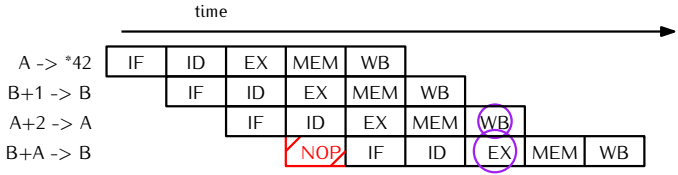
Pipeline hazards



1. Structural Hazard

Both `A->*42` and `B+A->B` require memory simultaneously. We need to **delay 2nd instruction**.

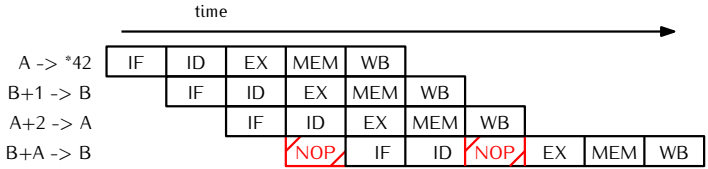
Pipeline hazards



2. Data Hazard

`B+A->B` can only be executed correctly when `A+2->A` is fully finished, ie we need to wait for the value of `A` to be fully updated.

Pipeline hazards



2. Data Hazard

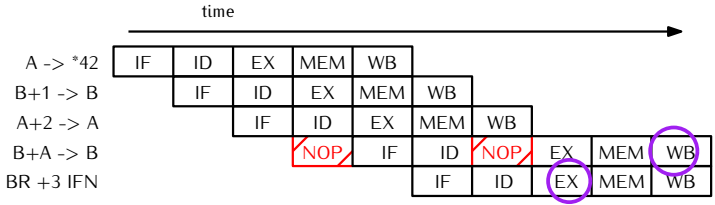
`B+A->B` can only be executed correctly when `A+2->A` is fully finished, ie we need to wait for the value of `A` to be fully updated.

Pipeline hazards

time →

A -> *42	IF	ID	EX	MEM	WB					
B+1 -> B		IF	ID	EX	MEM	WB				
A+2 -> A			IF	ID	EX	MEM	WB			
B+A -> B				NOP	IF	ID	NOP	EX	MEM	WB
BR +3 IFN						IF	ID	EX	MEM	WB

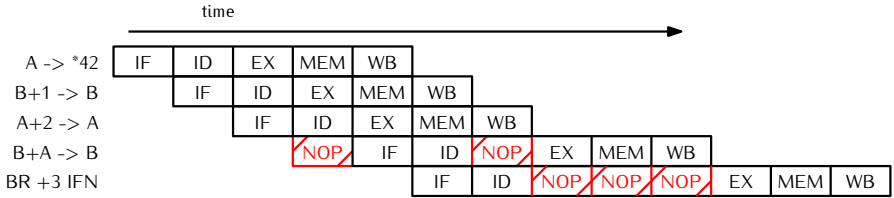
Pipeline hazards



3. Control Hazard

The control flow associated to instruction `BR +3 IFN` can only be evaluated when **B** is fully updated by previous instruction.

Pipeline hazards



3. Control Hazard

The control flow associated to instruction `BR +3 IFN` can only be evaluated when **B** is fully updated by previous instruction.